Massachusetts Institute of Technology
Lincoln Laboratory

A Study of Gaps in Cyber Defense Automation

George Baah
Thomas Hobson
Hamed Okhravi
Shannon Roberts
William Streilein
Sophia Yuditskaya

Technical Report 1194

October 18, 2015

This work is sponsored by the NSA's Cyber Defense Research & Technology Team under Air Force Contract #FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government. Distribution A: Public Release

Lexington Massachusetts

TABLE OF CONTENTS

This page intentionally left blank.

LIST OF FIGURES

This page intentionally left blank.

LIST OF TABLES

This page intentionally left blank.

1. EXECUTIVE SUMMARY

Cyber defense automation (CDA) refers to automated response and recovery from cyber attacks while still preserving a certain level of mission functionality. The vision of CDA research is to build self-healing, self-immunizing systems. Seven major components are necessary to achieve this vision: attack/vulnerability detection, attack/vulnerability analysis, impact blocking, recovery, vulnerability patching, system cleansing, and an optional active response component (e.g., deception or counter-attack). In this report, by reviewing the state of the art for each of these components, we identify high-priority, short-term research objectives for CDA components, which include: designing low false positive vulnerability detection techniques, developing scalable and fast-impact blocking mechanisms, accurately identifying the location of vulnerabilities, developing new roll-back techniques, evaluating various deception options, and using sanitization techniques for improved cleansing of compromised systems. These efforts will constitute the basic blocks of an effective and automated CDA system.

2. INTRODUCTION

Cyber defense has traditionally been an uphill battle. While defenders have to protect all components of a system, attackers have to find only one or a small number of vulnerabilities to successfully compromise the system. Moreover, the high cost of triaging attacks, developing patches, and cleaning up the damages often allows attackers to be one step ahead of the defenders. Most attacks also operate in the machine time scale (milliseconds), while defenders and operators work in the human time scale (hours/days), thereby impeding fast remediation of attack damages.

CDA can rebalance the landscape by making responses to cyber attacks faster, more seamless, and more automated. The vision of CDA is to create a self-healing, self-immunizing system that can detect, triage (analyze), block, and recover from cyber attacks. Moreover, such a system must be able to automatically develop patches to fix vulnerabilities, remove the presence (persistence) of the attacker from compromised systems, and provide misinformation/deception to the attacker.

In the current cyber landscape, there are vast challenges for achieving a self-healing, self-immunizing system. Major gaps in software and vulnerability analysis make the attack analysis and automated development of patches difficult. Moreover, because of the expressiveness and generality of modern systems, it is often possible to achieve the same malicious effect through various means. This makes blocking of attacks difficult to achieve. As a result, in the short and medium term, progress in this area is likely to be incremental.

2.1 GOALS

This report identifies the major gaps and challenges that currently exist in achieving the CDA vision. By analyzing the state of the art in CDA research, we identify and prioritize the important gaps that need to be researched in order to construct a self-healing, self-immunizing system.

The goal of this study is to find technical gaps within research projects that can be addressed in the short term to make effective headway in building a CDA system.

2.2 SCOPE

For this report, we focus our study on prominent cyber security research. To this end, we evaluate the publications in top-tier security conferences, including: IEEE Symposium on Security and Privacy (Oakland), ACM Computer and Communications Security (CCS), Network and Distributed Systems Security (NDSS), USENIX Security, Symposium on Research in Attacks, Intrusions, and Defenses (RAID), and Annual Computer Security Applications Conference (ACSAC). We have incorporated papers from other relevant conferences (including programming languages and operating systems conferences) as necessary.

Since cyber security research progresses very rapidly, we focus our study on the past four years, but include older publications if they constitute the state of the art in that area.

We identify the gaps in different components of a CDA system in Section 3, while excluding the areas covered in our previous reports. Most notably, we exclude the gaps related to attack detection and attack/vulnerability analysis as those have been covered in our “Attack Analysis” report [22]. However, we include the important gaps related to vulnerability detection, especially those closely related to a CDA system.

The rest of this report is organized as follows: Section 3 provides an overview of a CDA system and the components involved in building such a system; Section 4 describes our methodology, gap selection, and gap prioritization processes; Section 5 describes the state of the art for each of the CDA components and the high-level gaps; Section 6 describes the overall summary and prioritization of the gaps; Section 7 concludes the report.

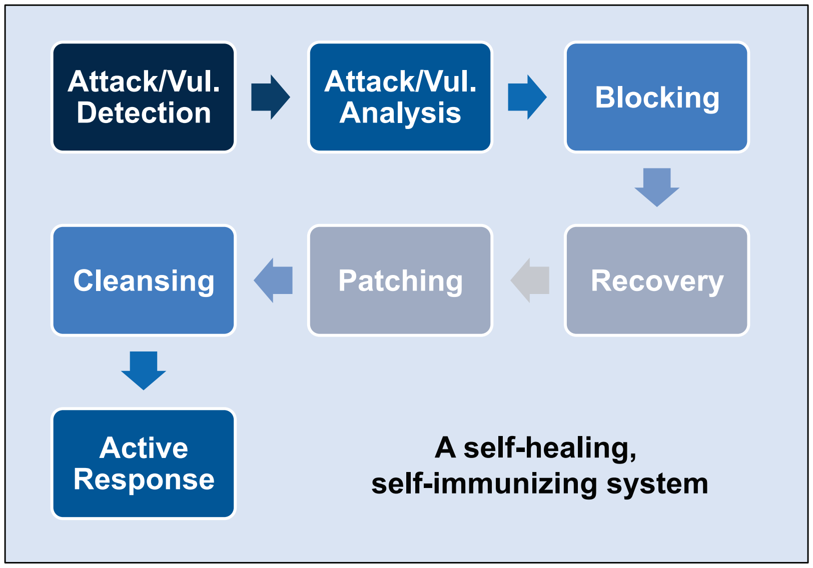

Figure 1. Components of a Cyber Defense Automation System

3. OVERVIEW OF A CYBER DEFENSE AUTOMATION SYSTEM

A Cyber Defense Automation (CDA) system is a set of technologies and capabilities that enable automated mitigation and remediation of vulnerabilities and active attacks on a network. Capabilities consist of activities and processes that support an intelligent, agile use of cyber defense information to form a self-healing, self-immunizing cyber system. The components of a CDA system include: Attack/Vulnerability Detection, Attack/Vulnerability Analysis, Blocking, Recovery, Patching, Cleansing, and Active Response, as shown in Figure 1.

An automated cyber defense response begins with Detection of an ongoing attack or an existing vulnerability in the network. Speed and accuracy of detection is important in order to take action to mitigate threats before they can do damage to network assets or disrupt missions. Examples of detection capabilities include intrusion detection systems [32], machine-learning analytics that can distinguish between suspicious and benign network activity, and automated fuzz testing techniques that can discover previously unknown vulnerabilities in software [5]. In this report, we focus on Vulnerability Detection, because Attack Detection was covered in our previous "Attack Analysis" report [22].

The Analysis component looks at the specific attributes of the attack or vulnerability to reason about its behavior and impact in the context of the prevailing threat model. This enables accurate, informed decisions that dictate mitigations appropriate to the nature of the attack and the missions it affects. Examples include behavioral analysis of malware [24] and analyzing traffic from bots to determine the C&C servers among them [11]. Please refer to our previous report, "Attack Analysis" [22], for details and gaps pertaining to automated Analysis techniques.

Having detected and analyzed an ongoing attack, the system should be able to take swift action to Block it while preserving a sufficient degree of service to minimize disruption to mission-critical operations. Blocking activities include attempts to stop or reduce the *impact* of an ongoing attack as well as the reachability of a vulnerability. In many cases, degraded service in the short term is inevitable; a CDA system must, however, be able to prevent a complete shutdown of the network. Activities in the Blocking category include isolating affected hosts or ports from the network, adding rules to a firewall to block traffic from known malicious domains, and techniques for automated enforcement and adjustment of access control and other security policies [16, 33].

Once an attack has been contained, the Recovery component of the system should take steps to roll back the network to its clean good pre-attack state, enabling the network to provide acceptable although possibly degraded service. In addition, Recovery can take proactive self-healing measures to fix instances of bad state on the network, even if they happen inadvertently due to human error or benign system failures. Examples of Recovery activities include rolling back a VM to a previous snapshot, detecting and fixing bugs in access control logic [29], and using rescue points to facilitate automated execution recovery in multitier architectures [37].

Another important part of a post-attack self-healing process is the Patching of vulnerabilities that made the attack possible. Patching can happen at two different time scales. At the machine time scale, patches can be developed automatically to quickly fix the vulnerability. At this scale, patches can be developed quickly, but they may be unreliable. Unreliability refers to possible disruption of the system's mission, failure to actually fix the vulnerability, and possibly introducing new vulnerabilities into the system. At the human time scale, patches are developed by human operators/developers to permanently fix a vulnerability. At this scale, patch development is slow (often weeks/months), but it is more reliable than automated patch development. A complete CDA system should incorporate both patching components for an effective remediation of attacks: automated patches are developed to apply a quick fix to the vulnerability before better, more stable patches are made available and applied by the vendors.

The detection, analysis, and patching of vulnerabilities need not wait for an attack to happen. In fact, it is preferable for these capabilities to strengthen the network's defenses proactively, to prevent attacks from occurring in the first place. An automated solution requires ongoing monitoring of the network's systems and services against a continuously changing landscape of security patches and software updates coming in from vendors, as well as the ability to automatically deploy and test patches across the various host configurations and platforms that exist on the network.

The self-healing of a network is not complete until it has been Cleansed of any remaining traces or foothold the attacker may have left behind. As part of an attack, malware can be injected into various devices on the network and left dormant until cyber defense responses to the current attack subside. Also, the attacker may have gained access to the network by learning credentials or other sensitive information (e.g., through phishing). If credentials are left unchanged or the backdoors are not removed from the compromised systems, the network remains open for the attacker to launch a new attack at a later time. The challenge of developing an automated and thorough Cleansing system is being able to detect these hidden backdoors, and to take sweeping prophylactic action, such as organization-wide password changes, with minimal disruption to the network and its missions.

Depending on the nature of the attack, it may also be desirable to launch Active Response operations against the attacker, which include deception tactics and direct counter-offensive measures. A deception capability can divert the attacker's attention away from sensitive areas of the network, provide misinformation, and/or prevent the attacker from learning about the defense capabilities of the network. Counter-offensive measures go even further by actively inflicting damage on the adversary's systems and data to disempower the attacker at the source. Such active response capabilities can require expensive additional overhead, so the benefits of deploying them in a Cyber Defense Automation solution must be carefully considered against the costs, in the context of the network's specific threat model. Examples include setting up honeypots to attract attackers away from sensitive areas of the network [1], attacking vulnerabilities in exploit kits to render them harmless [7], and sending poisoned feedback to botnets to reduce spam [31].

This page intentionally left blank.

4. METHODOLOGY

We have conducted a systematic review of the state of the art in the development of each of the seven CDA areas, covering promising research that has been done within the last four years. In the course of our review, we have identified a number of existing gaps that remain towards achieving a CDA system that is truly self-healing and self-immunizing.

4.1 GAP DISCOVERY PROCESS

For the purposes of this study, we focus on the open literature and publicly known defensive techniques. We reviewed papers and studies, mainly from the prominent security conferences such as the IEEE Symposium on Security and Privacy (Oakland), Network and Distributed Systems Security (NDSS), ACM Computers and Communications Security (CCS), USENIX Security Symposium, Annual Computer Security Applications Conference (ACSAC), and the Symposium on Research in Attacks, Intrusions, and Defenses (RAID).

We initially surveyed the titles and abstracts for the past four years of each conference's proceedings. We then down-selected this very large list of papers based on their relevance to CDA. We read the papers from this refined list and extracted any relevant gaps based upon our own experience, the context of the paper, and the stated research contribution.

4.2 GAP SELECTION PROCESS

The gap selection process was performed during team meetings and began with individual team members independently determining if a paper was worthy of carrying forward. After gaps from each paper were identified, a majority vote was then used to determine whether to keep each gap or not.

4.3 GAP TREATMENT AND CLASSIFICATION PROCESS

There are three major types of gaps in existing cyber defense:

- Technology Gap: an effective defense does not yet exist against a type of attack.
- Deployment Gap: defenses have been proposed or implemented against an attack type (open source or commercial), but all of the existing defenses have impracticalities that impede their adoption.
- Practice Gap: practical and effective defenses exist in the community, yet they are not adopted widely, perhaps because of the lack of awareness or incentives.

In this report, we primarily focus on technology gaps. Note, however, that the distinction between these areas is not well established and there are many gray areas. For example, knowing that an existing defense against an attack has high overhead (deployment gap) points to the fact that a faster defense must be developed against that attack type (technology gap). As another example, a defense may only focus on a specific set of issues, making it not widely deployed (practice gap) and at the same time, not completely effective (technology gap).

4.4 GAP PRIORITIZATION PROCESS

For gap prioritization, we take a two-step approach. First, to prioritize the CDA components, we analyze their dependencies and assign higher priorities to the component upon which many other components depend. This prioritization process is explained in greater detail in Section 6. Second, to prioritize the gaps within each component, we used a subjective ranking system. Each team member was assigned a CDA component, and as such, became an expert in that knowledge area. Team members then identified the highest-priority gaps in their respective research area based on the number of papers highlighting the gap, whether the gap has truly been solved, and whether current technology exists that can solve the problem. We then combined these subjective rankings to come up with an overall ranking for the gaps. Note that our priority ranking is a subjective computation based on the qualitative ranking of our team members. Other researchers are likely to compute different priorities based on their scoring of the gaps.

5. GAPS AND RESEARCH DIRECTIONS

5.1 GAPS IN VULNERABILITY DETECTION

5.1.1 High-Level Gaps

1. False positive rates are too high to be practical, especially for automated remediation techniques after vulnerability discovery.
2. Vulnerability triaging is a highly manual task, as human-in-the-loop expertise is required for assessing the threat posed by detected vulnerabilities and prioritizing them.
3. Vulnerability discovery techniques continue to grapple with the need for scalability and efficiency to keep up with changes in complex, dynamic systems.
4. Vulnerability detection tools are a boon to the adversary, enabling them to more easily and efficiently detect the presence of vulnerabilities.

5.1.2 Technique: CANVuS

Description of Technique

CANVuS [35] is a triggering event-driven solution to proactive network vulnerability scanning that is guided by changes in network context, such as hosts joining the network, reconfigurations, or changes to network services. Context-aware vulnerability detection approaches, such as the one implemented by CANVuS, reduce resource overhead and detection latency compared to polling-based models that indiscriminately launch periodic scans over the whole network.

Architecturally, CANVuS places network monitors in servers, routers, and switches across the network that look for infrastructure, service, and packet-based events of interest. A context manager aggregates the event data, translates it into a uniform model, and stores it in a network state vulnerability database. CANVuS makes use of existing sources of data (e.g., syslog, SNMP, and Netflow), while a flexible plug-in framework facilitates adding other data sources. The database keeps track of the results of each scan completed for every host, storing information such as the time the scan was triggered, a list of open ports, a list of vulnerabilities on each port, and the type of triggering event.

Operationally, CANVuS begins by querying the network state vulnerability database for all available hosts, which become the initial scanning candidates. A FIFO scheduling strategy is used to minimize network overhead due to scanning. Callback functions containing database triggers automatically append a new candidate to the scan queue whenever a relevant change happens to that host. Triggering events are configured as policies in CANVuS, according to the discretion of the network defender. Each host is also equipped with a timer that adds it to the scanning queue if no changes have occurred since its last scan. The timer is also configurable by defenders. Nessus is used for the actual scanning operations, modified to support this queuing design.

Technique-Specific Gaps

CANVuS suffers in several ways due to the difficulty of gaining local access to individual hosts on the target network for host-based monitoring. Because many configuration changes occur at the host level, they are not observable through network events. Due to this inability to perform host-based monitoring, no real ground truth was available to evaluate CANVuS's accuracy, and thus, the true positive rate could not be analyzed. It is also for this reason that timeout-based scanning is part of the CANVuS solution. Although the timeout period is customized to each host's network state, scanning history, and administrator policy decisions, it is a static parameter that reduces the context-aware, event-driven properties of CANVuS. In fact, the use of timeout-based scanning drove up the latency to such a degree (6 hours) that there was no significant improvement over pure polling-based scanning (7 hours). This performance fails to achieve the stated context-aware property where scans are triggered instantaneously in response to evidence of configuration changes on the network. Unless some level of host information is monitored, the timeout-based method cannot be completely replaced with the trigger-based approach. Thus, feasible solutions for host-based monitoring need to be explored and developed. In the meantime, an automated mechanism that dynamically adjusts the timeout period as each host's scanning history evolves may increase the effectiveness of the timeout-based system.

CANVuS is only as effective as the policies that specify when events are triggered, which CANVuS leaves for the defender to manually define. The highly dynamic nature of many networks (e.g., those where the use of mobile devices is widespread) and the adaptability of the adversary to utilize novel avenues of attack may necessitate at least some automation to support frequent, dynamic updates to the policies. Also, the need remains for risk and threat assessment methodologies that will guide the defender in coming up with comprehensive and effective policies that capture the potential vulnerabilities on the network and the current threat landscape, while minimizing disruption to missions. While the CANVuS methodology includes a study of temporal correlations between triggering events and observed network configuration changes to help determine effective triggering policies, it focuses on permanent changes, each of which is defined as being the only change on a given host within a 16-day period. Not only does this eliminate 95% of hosts, resulting in a lack of statistical significance in results, but this definition of permanence is arbitrary. As such, other definitions of permanence need to be explored. Further, ephemeral changes can be very relevant to vulnerability scanning and risk assessment, and should be included in future analyses pertaining to the definition of triggering events.

5.1.3 Technique: ReDeBug

Description Of Technique

ReDeBug [12] addresses the problem of code cloning that is especially prevalent in open-source software. When code is cloned, any vulnerabilities it contains are cloned as well. When a patch is released for a vulnerability, all of the relevant code clones need to be patched, not only the original code that led to the development of the patch. Given that there are millions of files that need to be checked, a solution must be efficient and scalable, both in terms of memory and speed. Although much prior work has been done in this area, it has relied on language parsing, which tends to be slow, cumbersome, and incomplete. ReDeBug is a novel approach that focuses on maximizing efficiency through the use of Bloom filters and feature hashing.

ReDeBug has two use cases: detecting unpatched code clones in a given file, and evaluating the amount of code cloning that is shared by two files (i.e., the similarity between them). In both applications, ReDeBug first normalizes each code file by removing redundant whitespace and converting all characters to lowercase. Next, the normalized file is tokenized using an n-length window. These n-tokens are then hashed into a Bloom filter for that file.

In the code clone detection use case, given a unified `diff` software patch, ReDeBug performs a Bloom filter set membership test to detect whether a code file contains the vulnerability that is fixed by the patch. In the second use case, to determine the amount of code cloning in common between two files, a similarity metric based on feature hashing is used to calculate the percentage of tokens in common between the two files. In both use cases, ReDeBug finishes with an exact match test on the identified unpatched code clones to remove Bloom filter errors.

Technique-Specific Gaps

One of the goals of ReDeBug was to provide a solution that can find a competitive number of code clones without requiring the use of clusters, which are expensive and not readily available to most developers. However, with a complexity of $O(N^2)$ where N is the number of files (or the number of functions if using function-level granularity), the code similarity detection algorithm is neither scalable nor efficient, and still requires a cluster to feasibly run. More efficient solutions to the similarity detection problem remain in need of development.

Some patches do not directly modify the vulnerable code; they insert a check before or after the vulnerable code. Such a patch will not be detected by the Bloom filter and this leads to false positives. The impact of erroneously patching a code clone that has already been fixed needs to be explored, especially for potentially mission-critical applications. Mechanisms for detecting such patches need to be developed.

A security implication of ReDeBug is that an attacker can find it very useful for malicious purposes. With a single laptop, an attacker can use ReDeBug to quickly find thousands of vulnerable applications among millions of files and billions of lines of open-source code. To stay ahead of the adversary within the sheer scale of the number of unpatched code clones that exist in open-source code bases (e.g., 15,546 unpatched code clones found in Debian and Ubuntu Linux operating system distributions), a solution that automates the fixing of these unpatched code clones is needed.

5.1.4 Technique: Detecting Vulnerable Websites (DVW)

Description of Technique

The technique of Detecting Vulnerable Websites (DVW) [30] aims to address the growing problem of webserver malware, which exploits vulnerabilities in outdated or unpatched versions of popular content management systems (CMS), such as Wordpress and Drupal. The attacker injects malicious code into a host running a web server to get it to participate in search-engine poisoning or redirection campaigns that may promote questionable services (e.g., counterfeits) or act as a delivery server for malware.

Prior work in detecting such threats has focused on reactive solutions that can automatically determine the maliciousness of a given website. The DVW technique takes a proactive approach that predicts future compromise by looking for legitimate websites that are vulnerable before they become malicious.

DVW takes a machine-learning approach by building decision tree models using a variety of features characterizing a website's traffic patterns, popularity rankings, file system structure, and non-user-generated webpage content. These features are prioritized by statistics that measure the impact of each feature on the website's chances of becoming compromised, and the top N features are submitted to the classification process. A dynamic feature extraction process periodically recomputes this set of top N features using a windowing and weighting strategy. Such dynamism is essential in order to capture the constantly evolving nature of this domain, as new CMS versions come out and old ones fall out of use.

Technique-Specific Gaps

Demonstrating a 66% true positive rate and a 17% false positive rate, DVW's best-performing classifier is not yet practical. Reliance on non-user-generated content and traffic features as the only indicators of a malicious website may be insufficient. In addition, DVW depends on blacklisted websites as ground truth for its training process, and is therefore limited to the accuracy of the blacklists themselves. More research is needed to explore additional feature sets, alternative feature selection statistics and sources of ground truth, as well as other windowing and weighting configurations, to improve DVW's classification performance.

The dynamic feature extraction system also has several limitations that demand more research. First, redundant features reduce the diversity and usefulness of the feature set. As such, correlation of top features to remove redundancy is suggested for future work. Secondly, there are scalability concerns that come with DVW's best-performing classification strategy, which takes into account all the past useful features along with the current top N features. Because the size of the feature set will therefore be monotonically increasing, training and running DVW will take longer over time. Further investigation is needed into methods that can achieve effective classification while keeping the size of the feature set relatively constant over time. A third limitation is that the adversary can poison the performance of the feature extraction system by adjusting the content on websites over which they have control. For example, if these websites have distinguishing tags, the adversary may remove them or rewrite them in semantically equivalent ways to prevent DVW from finding them useful for classification. The impact of this potential new threat deserves further analysis.

There are also concerns surrounding the use of a dictionary structure for training, which stores a list of all tags from all sites, mapping each one to a count of benign and malicious sites in which it has appeared. Unsurprisingly, this dictionary grows extremely quickly; for scalability, DVW adopted the heuristic of periodically purging all tags that had appeared only once during a given window (e.g., every 5000 sites added). This removed approximately 85% of the content from the dictionary every time it was run. The use of this particular heuristic poses potential problems due to the loss of valuable information, reducing DVW's ability to capture small changes in features over time and resulting in an increasing number of false positives. This also makes DVW vulnerable to adversarial poisoning of features. The actual impact of this heuristic needs further investigation. A consideration of alternate heuristics that may offer better trade-offs is also recommended. Also, the use of persistent storage (e.g., database) compression methods, as well as more memory-efficient representations for storing the tag-count mappings, remain to be explored, which may eliminate the need for such trade-offs entirely.

5.1.5 Technique: MAYHEM

Description of Technique

Mayhem [6] is a technique that attempts to automatically find and generate exploits for vulnerabilities in binaries. Mayhem relies on a hybrid combination of concrete and symbolic execution to find the vulnerabilities in a program. Given a program and an input description that specifies potential symbolic sources, Mayhem runs two processes: a concrete executor client (CEC), which concretely executes code on the CPU, and a symbolic executor server (SES), which can run on any platform. Mayhem is also more interested in attacker-controlled inputs (i.e., input sources that an attacker can manipulate).

When Mayhem loads a vulnerable program, it instruments the program and performs dynamic taint analysis while executing the program concretely. When execution reaches a branch or jump target that is tainted, Mayhem suspends the CEC engine's process and then sends the branch information to the SES engine for symbolic execution. The SES determines which branches are feasible and later sends the CEC the next feasible branch target to concretely execute. Note that during symbolic execution, SES forks a new execution each time the SES encounters a branch. SES maintains a path constraint for each forked execution. Forking executions lead to the path explosion problem that all symbolic execution engines encounter and this problem leads to the draining of system resources. To mitigate the path explosion problem, Mayhem suspends the execution of some forked executions until system resources become available for them to run. Mayhem also performs several other optimizations that allow it to analyze program binaries.

To automatically generate an exploit, Mayhem generates an exploitability formula when it detects a tainted branch. Mayhem queries a Satisfiability Modulo Theories (SMT) solver to determine whether the formula is satisfiable. If it is, then an exploit has been found. Mayhem's exploit generation technique is sound, but incomplete. It is sound because when Mayhem determines that a bug is exploitable, but it is incomplete because it is not able to find all exploitable bugs.

Technique-Specific Gaps

Although Mayhem is effective at automatically finding and generating exploits, it has a number of gaps. First, Mayhem can only generate exploits for some control flow hijack vulnerabilities. Application of techniques like Mayhem to other types of vulnerabilities remains an open research problem. Second, Mayhem cannot find all the vulnerabilities in binaries. This is because of the limitations inherent in symbolic execution. Third, Mayhem suffers from the path explosion problem experienced by all symbolic execution techniques, which makes it impractical for large code bases. Fourth, it requires significant manual effort to model the system and library calls. Fifth, Mayhem cannot handle multi-threaded programs. Further research is needed to address these gaps and tackle the scalability and manual overhead problems that have impeded the application of such techniques in production environments.

5.2 GAPS IN ATTACK/VULNERABILITY ANALYSIS

The attack and vulnerability analysis areas are covered in great depth in our “Attack Analysis” report [22]. For the sake of brevity, we do not repeat them here. Interested readers are encouraged to refer to that report for more information.

5.3 GAPS IN BLOCKING

5.3.1 High-Level Gaps

1. Automated blocking actions are unscalable, e.g., with symbolic execution/code analysis.
2. Blocking systems are still not fully automated, thereby requiring heavy manual effort.
3. Blocking techniques are still bypassable:
    - High-level policies (e.g., blocking capabilities) are easier to bypass.
    - Low-level policies (e.g., memory-instrumentation) are unscalable.
4. Blocking can be made ineffective depending on the system, i.e., they are limited by mission impact:
    - Dynamism of system/connectivity often reduces the impact of blocking.
    - Mission impact may limit blocking options.
    - Expressiveness of systems can make blocking ineffective.
5. Existing blocking techniques are overspecialized (very narrow in their applicability).

5.3.2 Technique: MetaSymploit

Description of Technique

Attack frameworks are tools that allow adversaries a method to rapidly create, configure, and deploy attack scripts. With attack frameworks, new attack scripts can be generated as soon as an exploitable vulnerability is discovered. This poses a challenge to administrators who are tasked with securing networks as they cannot generate or deploy mitigations at the same speed that attack scripts are generated [32].

MetaSymploit is a system designed to combat these zero-day vulnerabilities. In order to do this, MetaSymploit first symbolically executes an attack script. From the symbolic execution, the attack behavior and payloads are captured. Patterns and contexts are extracted from the attack script payload and behavior, respectively. From this, an IDS signature is created to block the attack script [32].

Technique-Specific Gaps

Though MetaSymploit sets out to achieve a lofty goal, it is overly specialized in its current implementation. As of now, it works for attack scripts generated from the attack framework MetaSploit, focuses on 10 specific attack script patterns, and is only valid for attack scripts written in current and past versions of Ruby. In addition, as each IDS signature is specific to each script, multiple versions of the same attack script will generate multiple signatures, thereby slowing down machine speed. Lastly, manual analysis is required when the attack script uses obfuscation techniques, complex symbolic loops, etc., or when multiple paths of an attack script lead to the same payload [32].

5.3.3 Technique: DEMACRO

Description of Technique

To deliver more enriched content to websites, platforms such as Adobe Flash and Microsoft Silverlight often utilize cross-domain requests. Cross-domain requests rely on configuration policies that state whether an applet is allowed to fetch content from a website's server. If a website's server has an insecure configuration policy, such as using wildcards, attackers can execute cross-domain attacks, e.g., leaking sensitive information or hijacking sessions [16].

DEMACRO aims to defend against these malicious cross-domain requests. It does so by first observing all requests created in a user's browser. If the request is caused by a plug-in applet (i.e., Adobe Flash) and is cross-domain, two tests are conducted to determine if the request is insecure: (1) does the website's configuration policy have insecure wildcards; and (2) does the applet expose proxy functionality. If either of these conditions is met, authentication information is removed from the request (i.e., from session cookies and authentication headers) and the request is allowed to complete. This ensures that such authentication information cannot be stolen by a cross-domain attack [16].

Technique-Specific Gaps

Similar to other techniques designed to combat malicious cross-domain requests, DEMACRO is an overspecialized tool. It only focuses on Adobe Flash plug-in based applets. In addition, DEMACRO only detects two types of misuse cases: insecure policies and insecure proxies. Recent reports estimate that only 11% of websites contain wildcard policies. Furthermore, during the DEMACRO evaluation phase, only 0.88% of all requests were deemed insecure. This suggests that DEMACRO is solving a small, unique problem that does not affect many websites. Lastly, DEMACRO removes authentication information contained in session cookies and headers. If private data is contained in other locations, it can still be leaked [16].

5.3.4 Technique: IceShield

Description of Technique

JavaScript, while allowing greater versatility and flexibility with respect to website design, allows adversaries to initiate a variety of web-based attacks, such as drive-by download and click-jacking attacks. Past mitigations to web-based attacks often introduce significant overhead or are not compatible with the current infrastructure [8].

IceShield distinguishes itself from past mitigation approaches by performing the analysis directly in the browser, thereby reducing overhead, and by being independent of the browser type. IceShield performs a dynamic analysis of websites and mitigates attacks by destroying the exploit's payload. It begins by overwriting and wrapping the JavaScript methods. Next, the called function and its parameters are inspected. The method is then compared to a threshold to determine whether it qualifies as malicious. The threshold is derived from machine learning (latent Dirichlet analysis), where a set of eight heuristics is used to distinguish benign and malicious websites. If the method qualifies as benign, the method is called with its original parameters, i.e., everything remains unchanged. If the method qualifies as malicious, the method is blocked or the set of arguments is modified [8].

Technique-Specific Gaps

IceShield is a tool built for a very specific purpose. It was designed to combat drive-by download attacks that match a specific set of heuristics, use native methods to prepare data structures, and deploy an exploit without redirecting or creating a new window context. In addition, IceShield cannot detect timing attacks, ActiveX attacks, or items that do not use DOM methods to create attributes, e.g., malicious PDFs or Java applets. Lastly, IceShield was evaluated against exploits created using a popular exploit kit. If an attacker crafts an attack that does not use an exploit kit or utilizes any sort of obfuscation technique, IceShield will not be able to detect the attack [8].

IceShield relies on the use of machine learning to determine its threshold. As such, it suffers from other issues associated with machine-learning approaches [22]: (1) it is only as good as the data it is trained on; (2) false positives require human analysis; and (3) features indicative of a malicious website may not necessitate maliciousness, e.g., a large header size may be identified by machine learning as malicious, even though a large header size can be caused by other benign reasons [8].

5.3.5 Technique: DeepDroid

Description of Technique

Many employees in modern enterprises use mobile devices to conduct business. As such, it is important for these devices to be managed and protected. Device administrators often want to restrict the use of specific applications while employees are at work, e.g., do not allow the use of microphones. At the same time, many applications only allow the user to choose from allowing all application permissions or not running the application at all [33].

DeepDroid was designed for corporate device administrators to have more control over mobile device applications via an enterprise security policy. With original equipment manufacturer (OEM) support, DeepDroid is installed on mobile devices with root privilege. The device is then authenticated to the enterprise policy center via a secret key. The enterprise policy center then distributes policy rules to the device. The device receives the policy and authorizes access privileges to individual applications. DeepDroid is a blocking technique because a new policy can be loaded on the devices based on attack/vulnerability detection events, quickly blocking their impact [33].

Technique-Specific Gaps

Though DeepDroid was designed for users of corporate mobile devices who may use their device for personal use, it cannot handle typical use cases. For example, if a user decides to install many applications, DeepDroid would require the creation of individual policies for each application describing what resources the application can access. In addition, DeepDroid cannot enforce policies on high-data-volume media-related resources such as the camera. With respect to policy creation, it is unclear where the responsibility lies. No matter the case, creation of individual policies for each application that a mobile device user decides to install requires significant manual effort. Lastly, there is always the chance that the mobile device user may deactivate the technology if it negatively impacts user experience, which is quite possible given that it causes performance overhead between 2 and 10%. All of these issues combined indicate that DeepDroid is not scalable [33].

5.3.6 Technique: JAMScript

Description of Technique

JAMScript is a program transformation technique that weaves a security policy into a JavaScript program so that the program is guaranteed to be safe with respect to the policy. The intuition behind the technique is that static analysis alone cannot prove that a program adheres to a given security policy, and runtime monitoring via inlined reference monitors may result in degraded performance and may be unsound. JAMScript attempts to overcome these limitations by using a hybrid technique that relies on transactional introspection, which both allows the effects of executions to be known a priori so that corrective measures can be taken, and allows transactional introspection of dynamically generated code [13].

Given a security policy, JAMScript first identifies statements in the program that may cause policy transitions. The information is used to rewrite the statements, which results in a transformed program, i.e., the transformed program has the security policy woven into it. The technique prevents the violation of the policy during program execution. For dynamically generated code, the technique uses statement redirection to determine whether the code violates the policy. During the execution of the program, if it appears that the policy is going to be violated, the technique suppresses the execution [13].

Technique-Specific Gaps

Although JAMScript is successful in preventing policy violations, it has a number of gaps. It requires manual policy specification, which can be prone to error and incomplete. Also, the overhead incurred because of the instrumentation in the programs can be large. Moreover, the policy should evolve concurrently as the program evolves, potentially making maintenance of the program more expensive [13].

5.4 GAPS IN RECOVERY

5.4.1 High-Level Gaps

1. True recovery techniques are rare, e.g.:
   - Rolling back to a previous state is not true recovery as it may cause information loss.
   - Many techniques falsely refer to blocking or cleansing as recovery.
2. Rolling back a system is an imperfect method, e.g.:
   - System functionality may break (reliability problem).
   - System may roll back to a long time before the attack or even after the attack (accuracy problem).
   - In distributed systems or systems with many dependencies, rolling back is difficult.
   - Rolling back incurs large overhead.
3. Automated testing of recovery effectiveness and side effects is still a challenge.
4. Many techniques shift manual effort from one area to another, i.e., instead of manually recovering the system, the defender must manually develop recovery rules and templates that will be used to automatically recover the system.

5.4.2 Technique: Airmid

Description of Technique

Large-scale attacks on mobile devices have yet to occur as applications (apps) are often manually removed from the app marketplace soon after malware is discovered. This method of manually removing apps will not suffice as the number of applications exponentially increases. As such, a more automated solution for finding malicious apps is needed [20].

Airmid aims to automatically identify and respond to malicious mobile applications based on network traffic. It does so by first identifying suspicious network traffic through the use of sensors, e.g., domain name service (DNS) blacklists. A centralized server then relays a secure message to the mobile device. Software on the mobile device inspects the kernel to identify which process is responsible for the suspicious traffic. Once the process is identified, the device initiates a remediation action, i.e., process termination, traffic filtering, application updating, mobile device updating, file removal, or factory reset [20].

Technique-Specific Gaps

Airmid focuses on the detection of malicious network activity, thereby ignoring malicious device activity. Airmid also requires network connectivity and only examines the activity of transmission control protocol (TCP) connections, not user datagram protocol (UDP) connections. During the evaluation phase, Airmid was able to detect and remedy only three kinds of malicious activity, i.e., leaking of private data, dialing of premium numbers, and participating in botnet activity. These three issues indicate that Airmid is a problem-specific technique. In addition to being problem specific, Airmid relies on already-established detection techniques that rely on network sensor data, e.g., DNS blacklists. As such, Airmid is only as good as its detection technique, i.e., if DNS blacklists do not contain a particular attack type, then Airmid cannot detect or remediate attacks of that type. Lastly, if Airmid initiates a factory reset of the mobile device in response to an attack, all user data may be lost, thereby resulting in an irreversible and, in the case of a false positive, an unnecessary action [20].

5.4.3 Technique: FixMeUp

Description of Technique

With any web application, it is important to have proper access control and ensure that only authorized users have access to sensitive operations or data. Most access-control policies are written from scratch, as there is no general framework for writing policies. As such, developers sometimes unknowingly insert inaccurate access-control policies or fail to identify methods in which access control can be bypassed [29].

FixMeUp is a tool that identifies and repairs access-control issues for web applications. It begins by receiving an access-control policy regarding sensitive operations, e.g., database queries or file deletions, and then turns this policy into a template. The template is then compared to every instance in which a sensitive operation is called to verify that the access-control logic matches the template. Repairs are generated when the logic does not match the template or when some of the logic is missing. After ensuring that the new logic and the template match, the transformed code is sent to the web application developer for actual implementation [29].

Technique-Specific Gaps

FixMeUp relies on being fed an accurate access-control policy, which requires significant developer effort in combination with policy inference tools. Even if a developer perfectly specifies the access-control policy, FixMeUp cannot handle specific language features, e.g., dynamic class loading or external side effects. In addition, FixMeUp corrects access-control issues as specified in the template, with no exceptions. This can lead to unwanted changes in the program if: (1) the template issues access control incorrectly; (2) the template only indicates one check for access-control, yet multiple checks meet the objective; or (3) the template requires an unwanted control or data dependency to be introduced [29].

5.4.4 Technique: Cascading Rescue Points (CRPs)

Description of Technique

Despite developers' best efforts, software is not error free. When an error occurs, a popular solution is to terminate the program. In an effort to implement less drastic techniques, software self-healing based on rescue points (RPs) has been proposed. The premise behind the idea is that software already contains code for handling errors and this same code can be used to handle unexpected errors [37].

RPs are functions that contain error-handling code. Cascading rescue points (CRPs) build on rescue points by applying the concept to multitier architectures. More specifically, as a thread enters a RP function, code is inserted that switches the thread into checkpointing mode. In checkpointing mode, the CPU state is saved and memory contents are logged. CRPs add an additional layer by notifying remote peers of this status change so they can begin their own checkpointing. If a thread in an RP function receives an error signal, the recovery process begins as memory contents are restored. In addition, remote peers also begin the recovery process [37].

Technique-Specific Gaps

CRPs are not a scalable solution for two reasons: (1) CRPs cannot handle multiple errors received in a short time; and (2) implementing CRPs requires that all applications have the ability to checkpoint and recover. CRPs are a problem-specific solution as they can only be implemented for applications that communicate through TCP connections. In addition, CRPs only work for architectures with an innate hierarchy; there is no guarantee that a globally consistent state will be maintained across processes. Lastly, the current implementation of CRPs is not yet fully practical due to overheads upwards of 70%, even when not checkpointing or recovering, and because of the potential for false positives that leads to memory corruption as erroneous information is restored [37].

5.5 GAPS IN PATCHING

5.5.1 High-Level Gaps

1. Automatically finding the location of the error in the buggy application (fault localization) is unscalable.
2. Effective analyses (static or dynamic) to determine the effects of patches on the functionality of the application (impact analysis) are lacking.
3. Regression test suites for automatically generated patches are often inadequate:
    - Patches may not fix vulnerabilities.
    - Patches may introduce new vulnerabilities.
4. Automated test generation (symbolic, random, or search-based) is difficult [23].
5. Automatic determination of whether a given application passes or fails a test case (i.e., oracle problem) is difficult.

5.5.2 Technique: PatchDroid

Description of Technique

Android is the most popular mobile device platform, and as such, it is important for patches to be quickly distributed. Devices sold directly by Google receive over-the-air updates. Other devices rely on their manufacturer to deliver updates, which may or may not be timely. In addition, most devices only receive updates from their manufacturers for the first 1–2 years of use; after this point, devices are no longer updated, leaving them prone to security vulnerabilities.

PatchDroid [17] was designed to distribute and apply patches to these legacy Android devices through the use of a dynamic, in-memory patching process. PatchDroid is a system that runs in the background on Android devices. It monitors all Android processes to determine whether they are in need of a patch by comparing them with a list of known patched vulnerabilities. Once a process in need of a patch is identified, the patch is injected. Patch injection occurs if, and only if, the patch is known to be reliable, i.e., does not cause the device to crash. Lastly, PatchDroid also monitors the mobile device for attacks on already patched processes.

Technique-Specific Gaps

PatchDroid relies on both on-device and cloud components. The on-device component is responsible for process monitoring and patch injection, among other things, while the cloud component contains the patch library. The connection between the on-device and cloud components is not encrypted, thereby allowing an adversary an avenue for attack. In comparison to patches

Google or device manufacturers distribute, PatchDroid relies on independent security analysts to write patches. Though the patches are tested to ensure that they do not cause the device to crash, they may still cause unintended effects, such as data corruption. Lastly, PatchDroid is an over-specialized technique with limited coverage. More specifically, PatchDroid is a system designed for privilege escalation vulnerabilities and user space, e.g., within user application, vulnerabilities. It only patches vulnerabilities in native binaries and Java software. If a vulnerability exist outside of this space, e.g., in manufacturer-specific portions of the operating system, PatchDroid cannot manage it.

5.5.3 Technique: Kali

Description of Technique

Kali [26] is a patching technique that was developed to address some of the limitations and evaluation issues of techniques that use genetic programming to find patches for buggy applications. To patch a buggy application, Kali applies a fault-localization algorithm to find the potential location of the error. The fault-localization algorithm uses the set of passing and failing executions and a score function to generate a ranked list of statements. For each statement, Kali searches the space of patches to find a patch that fixes that error. The patch search space of Kali consists of redirecting branches, inserting return statements, and removing statements. The authors show that using these types of patches performed better than the techniques that employ evolutionary algorithms, such as GenProg [34].

Technique-Specific Gaps

Although Kali performed better than other evolutionary algorithms, gaps still remain. First, Kali relies on three types of patches and therefore it cannot fix complex errors. Second, Kali does not perform any impact analysis to determine the effects of its fixes. For example, it is possible that the vulnerability still remains in the software despite the patch, or new vulnerabilities are introduced because of the patch. Third, Kali relies on the efficacy of the fault-localization algorithm to be successful and as such, a bad algorithm may prevent Kali from producing good patches or any patches at all. Finally, without a specification, it is impossible to determine whether the patch actually fixes the error in an application.

5.5.4 Technique: SemFix

Description of Technique

SemFix [21] is an automated patching technique based on symbolic execution, constraint solving, and program synthesis. To fix a software bug, SemFix relies on three existing techniques to generate a patch. The first technique is fault localization, which attempts to find the location of the fault in the software. The second technique is specification inference, which attempts to find the correct specification of the buggy statement. The third technique is program synthesis, which attempts to synthesize an expression that conforms to the specification discovered by the second technique.

Given a program and at least one failing test case, SemFix works by using statistical fault localization to generate a list of candidate program statements that may be buggy. The statistical fault-localization technique ranks the program statements from most suspicious to least suspicious. For each of the suspicious statements, SemFix generates a repair constraint using concolic execution (i.e., concrete and symbolic execution), which is then solved using a technique called component-based program synthesis. In component-based program synthesis, a function that satisfies a set of input-output pairs is synthesized from basic components. Components can be addition, subtraction, and constants.

Technique-Specific Gaps

There are a number of significant gaps that exist within SemFix. The first gap is that the technique only repairs buggy statements that are assignment ($x = f_{buggy}(\cdots)$) or conditional statements ($if(f_{buggy}(\cdots))$). Also, the technique cannot produce patches for bugs that span multiple statements. The second gap is that the technique can be inefficient given the large state space of functions that has to be searched to find a solution to a repair constraint. The third gap is that the patch generated by SemFix cannot be generalized to other test cases. This is because the repair constraint is generated based on the program's regression test suite. Any new test cases that are constructed for the program can render the patch invalid because the new test cases can invalidate the repair constraint. This gap is particularly severe because when a new patch is constructed for a given software, the new patch will introduce new program paths or data dependences in the program. The paths or data dependences must be tested, e.g., by developing new test cases. The fourth gap is that the technique is not able to generate patches for programs with large regression test suites. This is because large regression test suites generate large constraints, which become difficult to satisfy. The fifth gap is that the technique performs no impact analysis to determine the impact of the patch on the functionality of the program. Finally, the last gap is that the inaccuracies of the statistical fault-localization process may be misleading, causing the technique to spend large amounts of time fixing non-buggy statements.

5.5.5 Technique: AppSealer

Description of Technique

AppSealer [38] detects and automatically patches Android applications that are vulnerable to component hijacking. Component hijacking is where a malicious application that does not have privileges is able to use a vulnerable application that does have the desired privilege in order to retrieve or store sensitive data. They use static data flow analysis (backwards and forwards) to determine vulnerable code in apps that leaks sensitive information or stores user input in sensitive places. They insert instructions into the intermediate representation (IR) to track taint propagation at runtime as well as guard instructions at sinks in order to prevent dangerous data flow. The runtime taint tracking is used to eliminate false positives identified during the static data flow analysis.

Technique-Specific Gaps

AppSealer has a number of gaps. First, AppSealer patches only one class of vulnerabilities. Second, AppSealer does not perform any kind of impact analysis to determine the impact of the patch on the functionality of the application. Third, the technique suffers from false positives and false negatives, as determined by taint tracking. Fourth, AppSealer cannot perfectly translate from bytecode to IR, which can have a negative effect on its effectiveness. Finally, there is no guarantee that the patch that is generated is human-readable, impeding future software maintenance.

5.5.6 Technique: Code Phage (CP)

Description of Technique

Code Phage (CP) [28] patches vulnerabilities in stripped x86 applications by transferring checks from donor applications into the buggy recipient applications, thereby eliminating the error in the buggy applications. The intuition behind CP is that a donor application implements a check that handles the error-triggering input.

Given a benign input and an error-triggering input, CP searches a database of applications (i.e., donor applications) that correctly process the inputs. CP selects a donor application, instruments the application, and then runs the application on the inputs. At each branch condition that is influenced by the relevant input that triggers the error, CP records the direction taken by the branch and a symbolic expression for the branch. CP records the direction taken by the branch because it assumes that the benign and error-triggering input take different directions. Any of the branches where the benign and error-triggering input take different directions becomes a candidate check to be transferred to the recipient application. CP performs fault localization on the recipient application to determine where to insert the check. After the check is inserted, CP performs patch validation. During patch validation, CP ensures that the modified recipient application compiles correctly and passes all the regression test cases for the original recipient application.

Technique-Specific Gaps

Although CP was successful in patching a number of applications, it still has significant gaps. The first gap is that CP can only address three classes of errors: out-of-bounds access, integer overflow, and divide-by-zero errors. The second gap is related to the first in that CP can only address bugs that can be fixed with conditional checks. For example, divide-by-zero errors can be handled by checks that ensure that the divisor is not zero. In general, not all bugs can be fixed with conditional checks (e.g., race conditions). The third gap is that CP assumes that the error-triggering and benign input do not take the same path in the program. This assumption is not valid because there are bugs where the benign and error-triggering inputs take the same path on some inputs. The fourth gap is that CP assumes that there is a donor application that correctly handles the benign and error-trigger inputs, but that may not always be true. The fifth gap is that all of the checks end with an *exit(-1)* statement. This design may not always be desirable because exiting from the application is essentially a form of denial-of-service action. Finally, the sixth gap is that for every input that triggers a failure in the recipient application, CP will transfer a check from a donor into the recipient. In practice, this may lead to an application with code that is difficult to understand or maintain.

5.5.7 Technique: ClearView

Description of Technique

ClearView [25] is a technique for automatically patching stripped Windows x86 binaries. ClearView is designed to target heap-buffer-overflow and illegal-control-flow-transfer vulnerabilities. ClearView has five components that work together to generate a patch for a given binary. The first component is a learning component, which ClearView uses to perform dynamic invariant detection or specification mining. ClearView gathers invariants during the normal execution of the binary. Invariants are logical formulas whose values may be of registers or memory locations. After ClearView learns the invariants, it inserts monitors into the binary to detect heap-buffer-overflow and illegal-control-flow-transfer failures. When a failure is detected, ClearView attempts to find the invariant that is correlated with the failure. These correlated invariants are always satisfied during normal executions, but not in failing executions. ClearView uses the correlated invariants to generate a patch. The goal of the patch is to ensure that the invariant is not violated when execution reaches that program point. ClearView ensures that changing the flow of control, values of registers, and/or memory locations does not violate the invariant. It generates a number of patches, each of which is evaluated and the most effective patch is selected and applied to the binary.

Technique-Specific Gaps

Although ClearView was successful in generating patches for heap-buffer-overflow and illegal-control-flow-transfer vulnerabilities, it still has a number of gaps. First, ClearView targets only two classes of vulnerabilities, which limits is applicability to other classes of vulnerabilities or bugs that affect program functionality. Second, forcibly changing the control flow of a program, memory locations, or register values may have unintended consequences. ClearView has no way of identifying these unintended consequences because it does not perform impact analysis. Third, ClearView may not be applicable to large programs because it has a large learning overhead, which is due to the large number of invariants that it learns. Fourth, ClearView comes with a predefined set of invariants that it tracks. If an invariant needed to generate a patch for a vulnerability is not present, ClearView will not be able to fix that vulnerability. Fifth, ClearView assumes that the failure (observable error) occurs where the error in located in the binary. In general, the failure may occur far away from where the error is, thus preventing ClearView from generating a patch.

5.5.8 Technique: PAR

Description of Technique

PAR [15] is a technique that was developed to address the situation wherein automatic patch generation techniques produce nonsensical patches. Patch generation overcomes the production of bad patches by learning fix patterns from human-written patches. PAR then uses the learned fix patterns to automatically generate patches. The authors studied $62,656$ human-written patches of open-source projects and found that there several common fix patterns. An example of a common fix pattern is “changing a branch condition.” The authors created 10 fix templates PAR uses to generate patches.

To generate a patch for a buggy program, PAR uses an evolutionary-based approach. PAR first generates an initial population of program variants. Then, it modifies the initial program variants using the fix templates. PAR then uses a fitness function to select the top program variants. The fitness function measures the number of regression test cases that a program variant passes. These top program variants become candidates for the next iteration of PAR’s evolutionary process. The process continues until a program variant is found that passes all the regression test cases. For any given program variant, PAR does the following source code analysis: (1) PAR finds the location of the buggy statement in the program using statistical fault localization; (2) for a given buggy statement, PAR analyzes the statement and its surroundings in the program’s abstract syntax tree (AST) to determine which fix template to use to generate a patch; and (3) PAR rewrites the AST based on the template.

Technique-Specific Gaps

Despite the success of PAR in producing more human-readable patches, there are still significant gaps in the technique. First, inaccurate fault-localization results may lead the technique to be inefficient because it will spend time modifying non-buggy statements. Second, the technique does not perform any impact analysis to determine the impact of the patch on the functionality of the program. Third, new test cases need to be developed to test the impacted parts of the software. These new test cases also need to be classified as passing or failing test cases. Fourth, the technique relies on 10 fix patterns, limiting the types of bugs it can fix. Also, the technique cannot fix bugs that span multiple statements.

5.5.9 Technique: GenProg

Description of Technique

GenProg [34] is a technique that uses evolutionary algorithms to generate patches for buggy software. The key intuition behind GenProg is that a program that is missing a given functionality can recover that functionality in some other part of the same program. Given a buggy program and a failing test case, GenProg computes the abstract syntax tree (AST) of the program. In addition to the AST of the program, GenProg performs fault localization on the program to find statements that may potentially be buggy. The output of the fault localization is a weighted path in the program, that is, each statement has an assigned weight that determines how buggy it is. GenProg generates program variants by modifying only statements on the weighted path during its mutation and crossover operations. A mutation consists of inserting, deleting, or swapping a statement with another statement. A crossover consists of crossing over statements on the weight paths in two different variants. Also a fitness function is used to evaluate whether a given program variant makes it into the next generation. The fitness function gives more weight to variants that compile and pass more of the regression test cases. GenProg terminates when it finds a program variant that passes all the regression test cases.

Technique-Specific Gaps

Although GenProg can find fixes for certain faults, it has a number of gaps. First, the technique is expensive because of the large number of program variants it has to maintain. Second, because GenProg relies on the assumption that the correct implementation of a given buggy statement is located in some other part of the program, it will fail to find patches for buggy statements with no corresponding correct implementation. Third, the efficacy of GenProg is based on the effectiveness of its fault-localization technique. An ineffective fault localization technique will guide GenProg to other parts of the program that do not need modification and that may prevent GenProg from producing a patch. Fourth, the patches generated by GenProg can be nonsensical, as discovered by authors of the PAR [15] technique. Fifth, GenProg does not perform any kind of impact analysis to determine whether new test cases need to be developed to test the patch.

5.6 GAPS IN CLEANSING

5.6.1 High-Level Gaps

1. Effective cleansing requires a distinction between benign and malicious states, which has many of the challenges that exist in attack detection:
   (a) Keeping the benign state during the cleansing operation is difficult.
   (b) Removing the malicious modifications to the state during cleansing is difficult.
2. Complete cleansing requires the removal of the attacker's foothold from all system components (e.g., applications, operating system, firmwares, BIOS, etc.), which is unscalable and time consuming.
3. Verifying the correctness of cleansing is difficult.

5.6.2 Technique: Self-Cleansing Intrusion Tolerance (SCIT)

Description of Technique

The self-cleansing intrusion tolerance (SCIT) technique [2] is a technique that periodically rotates a virtual machine with its clean copy in order to decrease the exposure time of the system and remove attacker persistence. A separate system with a network attached memory stores persistent short-term information or session data between the systems. The final component is a controller that manages the rotation of the systems and how long each system copy is exposed. The systems can be in one of four states: the first state is active where it is online and accepting/handling requests; the second state is a grace period where it stops accepting new requests and finishes processing existing requests; the third state is inactive where it is taken offline to be restored; and the final state is a live spare where the system has been restored and is ready to become active. There can be either one active server at a time serving one service or multiple active servers serving multiple services. The latter would require additional algorithms to determine which systems could be easily brought down next. The systems are rotated on the order of minutes. SCIT has been applied to protect web servers [2], DNS [9], and other similar servers [10].

Technique-Specific Gaps

SCIT has a number of gaps and weaknesses. First, a technique like SCIT is best suited for servers that have to maintain minimal state and often serve non-user-changeable content (e.g., DNS). If the server state changes with user interactions, SCIT faces the difficult task of distinguishing between the benign state to keep and the malicious state to discard. Second, SCIT does not provide any mechanism for making such a distinction in the state that is kept in its network-attached memory. As such, it is possible that the attacker can persist in that state. Third, SCIT only removes attacker persistence from the components that are being wiped (i.e., the virtual machines). If an attacker can compromise the hypervisor or the controller system, it will not be effective. Fourth, there is no guarantee that the time spent on one server is not enough for attackers to achieve their malicious goal. As a point of fact, given the grace period, an attacker can prolong the presence on a compromised machine for an arbitrarily long period of time, invalidating the security guarantees of SCIT [2].

5.7 GAPS IN ACTIVE RESPONSE

5.7.1 High-Level Gaps

1. Active response is only appropriate for a limited class of attacks.
2. Achieving believability in both data and timing for stealth and deception continues to be a challenge.
3. Active response solutions are not fast or effective in preventing attacker success; existing techniques fall short of this goal.
4. Active response typically require higher system and human overheads than simply blocking an attack.
5. Current active response techniques require an expert human in the loop and customization to leverage their benefits.
6. Active response requires attribution with high confidence, which is an open problem.
7. Active response may be restricted by rules, regulations, and laws, while the attacks are not limited by such restrictions.

5.7.2 Technique: Honey-Patching

Description of Technique

Although critically important for cyber defense, patching has a significant drawback once applied: advertising to the attacker that a targeted vulnerability has been patched. For example, if a malicious request would have produced garbage output prior to patching, but now yields an error message, the attacker learns that the targeted host and vulnerability have been patched. This information provides many benefits to attackers. By being able to distinguish between patched and unpatched systems, attackers can quickly narrow down the pool of target hosts to maximize chances of attack success, meanwhile increasing their own confidence in the expected results of their attacks.

Honey-patches [1] frustrate attackers' ability to determine whether their attacks have succeeded or failed, by pretending that a patched machine is unpatched. It then redirects the attacker to a decoy honey-server that allows the attack to proceed with apparent success in an isolated environment. This redirection is achieved by adding a forking mechanism to the original patch that makes use of Linux's checkpoint and restore functionality to clone the target container in user-space. Efficient in-memory redaction cleans the cloned decoy of secrets and replaces them with fabricated, but realistic, honey-data. Defenders can gain valuable threat insights by instrumenting the honey-server with aggressive software monitoring to collect forensic information about the attack and the attacker. In addition, honey-server content can include false information that deceives and misinforms the attacker about the targeted host and network environment. Together, these capabilities greatly increase risk of detection and workload for the attacker.

Technique-Specific Gaps

This technique currently requires an ongoing process of manual development by someone with expert knowledge, as new patches come in and new applications and services are introduced into the network. While the implementation of forking is trivial for many kinds of patches, it is not so straightforward for patches that introduce deep changes to the application's control-flow or data structures. Transforming such patches into honey-patches requires correspondingly deeper knowledge of the patch's semantics. Further, honey-patching is inadvisable entirely for patches that close vulnerabilities by adding new, legitimate software functionalities, since those new functionalities increase the attack surface. It is recommended that honey-patching be applied judiciously based on an assessment of attacker and defender risk. Future work should consider how to reliably conduct such assessments [1].

Honey-patching is effective only against certain types of attacks. For example, it the attacker's goal is to exfiltrate sensitive data by exploiting a vulnerability, honey-patching can be effective. On the other hand, if the attacker's goal is to control the machine as a bot in a larger botnet, the honey container still satisfies the attacker goal, in which case honey-patching is ineffective.

The redaction process also involves a nontrivial manual implementation process that would best be done by a subject matter expert to minimize the risk of inadvertently leaving sensitive information accessible to the attacker in the honey-server. In addition, the honey-data that goes into the redaction process must be designed to be sufficiently realistic to deceive the attacker into believing that the honey-server is the target host instance. Because every server application has different forms of sensitive data stored in different ways, the redaction process must be specialized to each server product and its constituent applications prior to deployment, thus making redaction very unscalable.

Given this current semi-manual approach to honey-patching, automating the process is a greatly desirable next step, e.g., by incorporating it into a rewriting tool or compiler. The question of how to validate and/or audit the redaction process for arbitrary software is a challenge that remains to be addressed.

Operationally, because forking launches a new process in the form of a Linux container, it can expose the system to Denial of Service attacks. If attackers suspect or wish to find out whether honey-patching is in place, they could launch their attacks targeting vulnerabilities of interest a massive number of times in parallel until the physical target host becomes overwhelmed with (1) the number of spawned processes, and/or (2) memory and processor load due to cloning and redaction.

Thus far, honey-patching has only been developed and tested in the context of the Linux operating system, with the assumption that the attacker's targeted host instance exists in the form of a virtualized Linux Container (LXC). While Windows and Mac OS also provide checkpoint and restore capabilities, the effectiveness of the proposed honey-patching methodology remains to be evaluated on these platforms. Windows Server Containers are currently in development by

Microsoft as a parallel to Linux Containers. Expected to be released with Windows Server 2016, containers on the Windows platform may not be adopted for widespread use for some time [36].

5.7.3 Technique: EKHunter

Description of Technique

EKHunter [7] is a system that can arm the cybercrime analysts with powerful capabilities for counter-offensive operations against exploit kits (EKs). Its functionality includes: discovering and identifying exploit kits that may be present on a host or website, automatically analyzing the EKs server-side source code for vulnerabilities (if available, non-obfuscated, and non-object-oriented), and generating exploits that leverage these vulnerabilities. These exploits are then provided as a toolbox of capabilities that the analyst can choose and configure to suit the investigation and operational needs. Generated exploits can enable the analyst to do things such as initiate the takedown of an EK, render the EK harmless, gather intelligence about the EK's activities, and deceive the EK admin with false or arbitrary data.

Once the presence of an exploit kit on a given website is verified, EKHunter identifies the family to which this exploit kit belongs by leveraging the structural and behavioral signatures of exploit kits that it has previously encountered. To generate exploits for this EK, EKHunter makes use of multiple existing complementary and high-resolution vulnerability analysis tools to uncover access control flaws, SQL injection exploits, and various taint-style vulnerabilities in exploit kit code (e.g., XSS, file manipulation, and command injection). It then analyzes the vulnerable server-side code to automatically identify and retrieve all conditional statements and other constraints along each path leading to the vulnerability. Z3, an existing constraint solver, is used to construct an exploit string that can then be launched as part of an HTTP request. These capabilities are deployed as part of an exploit execution system that can be customized by the analyst.

Technique-Specific Gaps

EKs can easily evade EKHunter if they are developed with object-oriented structure, obfuscated, encrypted, or deployed as binaries. Partly because of this limited scope, the authors of this technique were only able to synthesize exploits for 6 out of 16 EKs that exhibited vulnerabilities in their proof-of-concept study. The need remains for similar work that addresses obfuscated EK code and binaries. Mechanisms for discovering vulnerabilities directly in malware binaries [4] may offer potential solutions in this area.

EKHunter can also be evaded if adversaries patch its targeted vulnerabilities in their EK code. The vulnerabilities that EKHunter currently looks for are predetermined and static. Focus thus far has been on relatively low-hanging fruit that is easily patched, such as SQL Injection and File Manipulation vulnerabilities. EKs are developed by a growing underground software development industry – as this becomes more sophisticated, and knowledge of EKHunter spreads, EKs will start to get patched and their code cleaned up. While this is already a win for defenders by raising the bar for the adversary, additional work is still needed to enable defenders to adapt their use of EKHunter to the changing EK code landscape in a timely manner to stay ahead of the adversary.

A system for tracking the versioning and patching status of EKs would enable defenders to stay current on the state of resiliency in specific EKs, so that they can focus their EKHunter operations on unpatched EK vulnerabilities. In addition, a mechanism for blocking the patching of already-deployed EKs and blocking new deployments of patched versions may be of interest to fill this gap.

5.7.4 Technique: B@bel

Description of Technique

B@bel [31] is a mechanism for filtering spam by distinguishing between simple mail transfer protocol (SMTP) conversations originating in botnets and those coming from legitimate email senders. This distinction is then used to send poisoned responses – falsely indicating that the recipient email address does not exist – back to spammers, forcing them to permanently remove the email address from their list or otherwise face unbounded performance penalties. B@bel is intended as a lightweight first step for spam mitigation to reduce the load on resource-intensive content analysis systems.

B@bel's techniques are based on the observation that there is wide variation in SMTP implementations across email clients. In particular, malicious clients are intentionally developed to ignore many SMTP standards of syntax and communication exchange in order to maximize the efficiency of botnet-based spam campaigns. For this reason, B@bel takes a machine-learning approach that uses decision state machines to build classifiers that can differentiate between various SMTP "dialects" and identify the dialect of any incoming SMTP conversation. This dialect identification is then leveraged to identify whether an incoming conversation is from a spambot. B@bel is then applied to filter incoming spam and respond with poisoned feedback to permanently eliminate any further spam from being sent by the originating botmaster to the affected email address.

Technique-Specific Gaps

In practice, B@bel has had limited success, demonstrating high false negative rates (21%) and providing only a 3.9% overall reduction in spam. Further, feedback poisoning has been effective in permanently eliminating spam from only 5 out of 29 spam campaigns. Combining this approach with TCP-level features [3, 14] or other attributes of email conversations could yield better results.

Spammers can evade B@bel by using legitimate mail transfer methods to deliver their spam messages (e.g., Sendmail, Postfix, Exchange), or by taking control of legitimate email accounts. More work is needed to address spam sent via these channels and clarify the extent to which spammers use legitimate SMTP clients as opposed to SMTP engines custom built for spamming efficiency.

Although B@bel appears to scale to hundreds of thousands of email conversations, a formal analysis of the performance overhead and scalability of this technique has yet to be done.

5.7.5 Technique: Beheading Hydras

Description of Technique

Beheading Hydras [18] addresses the problem of counter-offensive operations against botnets. The complexity, agility, and resilience of these malicious networks require highly coordinated takedown efforts involving law enforcement, security operators, and domain registrars, often across multiple countries. In most cases, takedown methodologies have been ad-hoc and limited by a lack of knowledge about the behavior of each botnet and its malware. Moreover, recent takedowns have resulted in collateral damage, bringing to light the need for a rigorous approach and tools to facilitate these complex, high-risk operations.

The Beheading Hydras technique consists of a takedown analysis and recommendation system, called *rza*, that helps analysts plan out effective takedowns of DNS-based command-and-control (C2) botnets. *Rza* takes in passive DNS (pDNS) and malware data to provide two functionalities: (1) enabling a post-mortem analysis of past botnet takedowns, and (2) providing recommendations on how to successfully execute future takedowns of similar botnets. The pDNS data consists of mappings between domain names and internet protocol (IP) addresses, constructed from real-world DNS resolutions observed in a large North American internet service provider (ISP). The malware data, collected from internal dynamic malware analysis output as well as commercial malware feeds, maps each malware sample's MD5 sum and binary to the domain names and IP addresses that it has queried.

Taking into account queries by known malware, shared infrastructure, as well as domain reputation, *rza* assembles a set of domains that are likely to be malicious. A date range is used to filter away historic relationships that may be outdated or no longer relevant. Domains that are in the set of Alexa top 10,000 are also filtered out, because they are unlikely to be persistently malicious and should not be considered for takedown. A malware analysis system [19] is also used to identify any contingency behaviors that the botnet includes in its malware, such as peer-to-peer (P2P) structure or domain name generation algorithms (DGA), that would allow the botnet to continue to function even after its primary C&C infrastructure has been disabled. The postmortem analysis component provides metrics for quantifying takedown improvement and estimating the potential risk of collateral damage.

Technique-Specific Gaps

While *rza* can identify the presence of P2P and DGA contingency behaviors, it does not offer analysis or recommendations for takedowns of botnets that employ them. P2P and DGA-based botnets use forms of nondeterministic behavior to their advantage, such as the randomness seed in DGAs and the peer enumeration and selection algorithms in P2Ps, which makes them far more resilient to takedown attempts than centralized C&C botnets. Related work in assessing the properties and vulnerabilities of P2P botnets [27] may be of service towards developing counter-offensive strategies and automated active response solutions in this area.

The use of a date range to remove outdated historic relationships raises the question of how to choose an optimal timeframe that does not miss valuable information. Such considerations must take into account the possibility of dormant botnets or those that are strategically timed. The choice of timeframe is perhaps best made on a case-by-case basis, given the particular history of each individual botnet. Alternate filtering mechanisms that can correlate more reliably with outdated relationships remain to be explored.

Another persisting challenge is that ground truth is difficult to obtain, which limits the ability to evaluate the accuracy of *rza*'s takedown recommendation engine, as well as the effectiveness of its risk and improvement metrics.

Even though *rza* provides analysis that supports counter-offensive operations, it leaves the actual decision making and takedown activities to the human-in-the-loop takedown teams. A centralized platform for coordinating takedowns would greatly facilitate takedown operations by these teams. Automated active response systems for botnet takedowns are yet to be developed; *rza* may offer insights and capabilities that can inform the design of such systems.

6. SUMMARY OF RESEARCH DIRECTIONS AND PRIORITIZATION

To develop a gap prioritization, we have used a two-step process. First, to prioritize among different components of a CDA system, we analyzed the dependencies among such components. Components that are at the root of the dependency tree, i.e., that many other components depend on, have higher priorities for research because without addressing their major gaps, there is little hope in effectively deploying other dependent components. For example, blocking depends on successful detection of attacks or vulnerabilities. Figure 2 illustrates the dependency graph for the CDA components. Note that cleansing does not depend on other components as it can be performed periodically, even without successful detection of an attack.

Second, in order to prioritize the gaps within a component, each team member voted based on her/his expert knowledge on what gaps should be ranked high, medium, and low. Although these rankings did not always match exactly, there was majority agreement over the priorities. In order to develop the final priorities, we met, discussed, and debated various technical and practical justifications for the rankings. We attempted to identify the major hindrances in each component that have traditionally prevented its widespread deployment in real systems. In some cases, many techniques exist for a component (e.g., detection), but they all share a common weakness (e.g., high false positive rates) that make them inappropriate for CDA systems. In other cases, implementing an effective technique for a CDA component relies on solving an open problem that has impeded significant advances in that area. For example, analyzing the side effects of an automatically generated patch relies on accurate code equivalence analysis, which is a known open problem in software analysis. In the discussion of the research directions, we also provide the major justifications for the research priorities for each CDA component.

Table 1 enumerates the gaps and research priorities for various CDA components. In the short term, we recommend the following research effort for high-priority gaps.

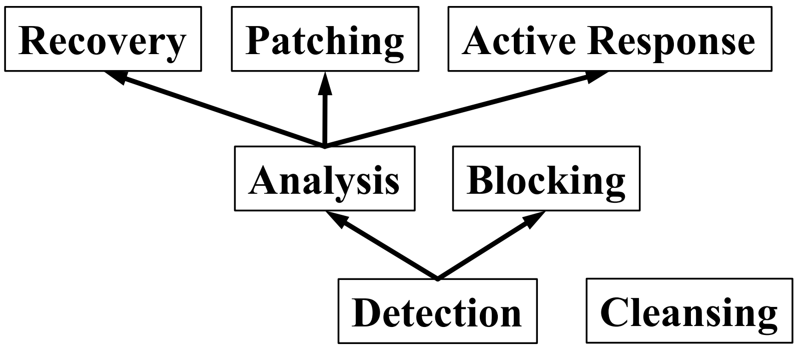

Figure 2. CDA Component Dependencies

For the vulnerability detection component, we recommend that research efforts focus on designing techniques with low false positive rates. Our justification for choosing this as a high-priority research area is based on the fact that although numerous vulnerability detection techniques exist, they all suffer from high false positives. This is a fatal weakness for CDA systems because automated responses based on a false positive are undesirable and will lead to operators who ignore the detection system. As a result, we believe that addressing the gap of high false positives and the necessary manual effort has the highest priorities in this area. Moreover, efforts should also focus on faster vulnerability triaging techniques that can analyze the impact and priority of a vulnerability in an automated fashion.

For the blocking component, we recommend that research efforts focus on developing scalable, automated mechanisms. Developing scalable and automated solutions was chosen as a high priority given that most modern systems are complex and integrated with other systems; as such, any blocking technique that does not allow or plan for this is unacceptable. Since blocking is one of the first stages in a comprehensive response, it may be acceptable if the mechanism is not complete or if it results in mission degradation. It is, however, crucial that the blocking mechanism be automated and applicable to large networks, as those goals constitute the main motivation behind the existence of a blocking mechanism.

For the recovery component, given that current techniques often refer to blocking or cleansing as recovery, we recommend that research efforts focus on developing new recovery techniques for different parts of a system (e.g., software applications, databases, and operating systems). In this way, true recovery across an entire platform can be achieved. Moreover, we recommend that high priority be given to reliable and accurate recovery techniques in a single system, before such techniques can be applied to broader distributed systems. Since some grace period is often allocated for recovery, it may be acceptable if some techniques in this domain have an overhead that is larger than ideal.

For the patching component, given that finding the location of code error is difficult across most applications, we recommend that research focus on optimizing fault (e.g., vulnerability) localization techniques. In addition, given that most patches often have unknown side effects, research examining new automated test development techniques that can better assess the effectiveness and side effects of automatically developed patches are of crucial importance.

Though there are few published papers focused on cleansing, as most servers have changing content, distinguishing between benign and malicious states is difficult. As such, we recommend that research focus on proper sanitization techniques to distinguish benign and malicious states. Data provenance techniques may be applicable in this area, as they can potentially identify all benign data on a system.

For the active response component, it is important to consider that the attacker is a human, and can easily detect deception or decoy methods. As such, we recommend that short-term research focuses on evaluating the believability of existing deception and decoy techniques. Although many deception techniques have been proposed in the literature, little work has been done on evaluating their believability. Moreover, research should also focus on identifying which classes of attacks can benefit from deception/active response. More specifically, although known examples exist for each class, little has been done on classifying applicable and non-applicable attack classes for active response. This distinction is important to ensure that techniques are not deployed with the assumption that they are a one-size-fits-all approach.

TABLE 1

CDA Component Priorities

Component	Gap	Priority
Detection	(1) Has high false positive rates	H
	(2) Is mostly manual	H
	(3) Is not scalable or efficient	M
	(4) Can be used by attackers	L
Blocking	(1) Is unscalable	H
	(2) Requires heavy manual effort	H
	(3) Is often bypassable	M
	(4) Can be limited by mission impact	M
	(5) Is overspecialized	L
Recovery	(1) Is rare	H
	(2) Is imperfect (e.g., lacks reliability and accuracy)	H
	(3) Is difficult to test effectiveness and side effects	M
	(4) Requires manual effort to build templates	L
Patching	(1) Is unscalable	H
	(2) Has unknown functionality impact (e.g., little is known about the effectiveness and side effects of patches)	H
	(3) Has inadequate testing mechanisms	M
	(4) Is difficult to automatically generate tests	M
	(5) Is difficult to automatically determine pass/fail of test cases	M
Cleansing	(1) Is difficult to distinguish between benign and malicious	H
	(2) Is difficult to accomplish completely	M
	(3) Is difficult to verify correctness	M
Active Response	(1) Applies to limited classes of attacks	H
	(2) Is difficult to make believable	H
	(3) Is not fast and effective	M
	(4) Requires heavy manual effort	M
	(5) Requires a human-in-the-loop and customization	M
	(6) Requires attribution	L
	(7) Is restricted by regulations	L

7. CONCLUSION

In this report, we have identified the major gaps in building a cyber defense automation (CDA) system. The vision of a CDA system is to achieve a self-healing, self-immunizing system that can provide fast responses to mitigate the immediate impact of attacks and recover to full functionality. Such a system will include seven major components: attack/vulnerability detection, attack/vulnerability analysis, impact blocking, recovery, vulnerability patching, system cleansing, and optionally, an active response component (e.g., deception or counter-attack). We identified the high priority gaps in building a CDA system by reviewing the state-of-the-art techniques proposed or built in this domain, extracting their implicit or explicit gaps and weaknesses, and raking those gaps based on the dependencies that exist among CDA components and the subject mater expert knowledge of the fundamental research problems. We believe that by focusing the short-term research on the high priority gaps, the community can make major headway in achieving the vision of a self-healing, self-immunizing system.

This page intentionally left blank.

REFERENCES

[1] Frederico Araujo, Kevin W. Hamlen, Sebastian Biedermann, and Stefan Katzenbeisser. From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 942–953, New York, NY, USA, 2014. ACM.

[2] Anantha K Bangalore and Arun K Sood. Securing web servers using self cleansing intrusion tolerance (scit). In *Dependability, 2009. DEPEND'09. Second International Conference on*, pages 60–65. IEEE, 2009.

[3] Robert Beverly and Karen R. Sollins. Exploiting transport-level characteristics of spam. In *Collaboration, Electronic Messaging, Anti-Abuse, and Spam Conference*, CEAS '08, 2008.

[4] Juan Caballero, Pongsin Poosankam, Stephen McCamant, Domagoj Babi ć, and Dawn Song. Input generation via decomposition and re-stitching: Finding bugs in malware. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 413–425, New York, NY, USA, 2010. ACM.

[5] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 380–394, Washington, DC, USA, 2012. IEEE Computer Society.

[6] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 380–394, Washington, DC, USA, 2012. IEEE Computer Society.

[7] Birhanu Eshete, Abeer Alhuzali, Maliheh Monshizadeh, Phillip Porras, VN Venkatakrishnan, and Vinod Yegneswaran. Ekhunter: A counter-offensive toolkit for exploit kit infiltration. In *Proc. 22nd Annual Network and Distributed System Security Symposium (NDSS'15). The Internet Society*, 2015.

[8] Mario Heiderich, Tilman Frosch, and Thorsten Holz. *IceShield: Detection and Mitigation of Malicious Websites with a Frozen DOM*, volume 6961, pages 281–300. Springer Berlin Heidelberg, 2011.

[9] Yih Huang, David Arsenault, and Arun Sood. Incorruptible self-cleansing intrusion tolerance and its application to dns security. *Journal of Networks*, 1(5):21–30, 2006.

[10] Yih Huang and Arun Sood. Self-cleansing systems for intrusion containment. In *Proceedings of workshop on self-healing, adaptive, and self-managed systems (SHAMAN)*, 2002.

[11] Gregoire Jacob, Ralf Hund, Christopher Kruegel, and Thorsten Holz. Jackstraws: Picking command and control connections from bot traffic. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 29–29, Berkeley, CA, USA, 2011. USENIX Association.

[12] Jiyong Jang, Abeer Agrawal, and David Brumley. Redebug: Finding unpatched code clones in entire OS distributions. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 48–62, 2012.

[13] Richard Joiner, Thomas Reps, Somesh Jha, Mohan Dhawan, and Vinod Ganapathy. Efficient runtime-enforcement techniques for policy weaving. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 224–234, New York, NY, USA, 2014. ACM.

[14] Georgios Kakavelakis, Robert Beverly, and Joel Young. Auto-learning of smtp tcp transport-layer features for spam and abusive message detection. In *Proceedings of the 25th International Conference on Large Installation System Administration*, LISA'11, pages 18–18, Berkeley, CA, USA, 2011. USENIX Association.

[15] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 802–811, Piscataway, NJ, USA, 2013. IEEE Press.

[16] Sebastian Lekies, Nick Nikiforakis, Walter Tighzert, Frank Piessens, and Martin Johns. *DEMACRO: Defense against Malicious Cross-Domain Requests*, volume 7462, pages 254–273. Springer Berlin Heidelberg, 2012.

[17] Collin Mulliner, Jon Oberheide, William Robertson, and Engin Kirda. Patchdroid: Scalable third-party security patches for android devices. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 259–268, New York, NY, USA, 2013. ACM.

[18] Yacin Nadji, Manos Antonakakis, Roberto Perdisci, David Dagon, and Wenke Lee. Beheading hydras: Performing effective botnet takedowns. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 121–132, New York, NY, USA, 2013. ACM.

[19] Yacin Nadji, Manos Antonakakis, Roberto Perdisci, and Wenke Lee. Understanding the prevalence and use of alternative plans in malware with network games. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 1–10, New York, NY, USA, 2011. ACM.

[20] Yacin Nadji, Jonathon Giffin, and Patrick Traynor. Automated remote repair for mobile malware. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 413–422, New York, NY, USA, 2011. ACM.

[21] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.

[22] Hamed Okhravi, Thomas Hobson, Chad Meiners, and William Streilein. A study of gaps in attack analysis. Technical report, MIT Lincoln Laboratory, 2014.

[23] Alessandro Orso and Gregg Rothermel. Software testing: A research travelogue (2000&#8211;2014). In *Proceedings of the on Future of Software Engineering*, FOSE 2014, pages 117–132, New York, NY, USA, 2014. ACM.

[24] Sirinda Palahan, Domagoj Babic, Swarat Chaudhuri, and Daniel Kifer. Extraction of statistically significant malware behaviors. In *Annual Computer Security Applications Conference, ACSAC '13, New Orleans, LA, USA, December 9-13, 2013*, pages 69–78, 2013.

[25] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 87–102, New York, NY, USA, 2009. ACM.

[26] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA 2015)*, 2015.

[27] Christian Rossow, Dennis Andriesse, Tillmann Werner, Brett Stone-Gross, Daniel Plohmann, Christian J. Dietrich, and Herbert Bos. Sok: P2pwned - modeling and evaluating the resilience of peer-to-peer botnets. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 97–111, Washington, DC, USA, 2013. IEEE Computer Society.

[28] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin. Automatic error elimination by horizontal code transfer across multiple applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2015.

[29] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. Fix me up: Repairing access-control bugs in web applications. In *NDSS*, 2013.

[30] Kyle Soska and Nicolas Christin. Automatically detecting vulnerable websites before they turn malicious. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 625–640, San Diego, CA, August 2014. USENIX Association.

[31] Gianluca Stringhini, Manuel Egele, Apostolis Zarras, Thorsten Holz, Christopher Kruegel, and Giovanni Vigna. B@bel: Leveraging email delivery for spam mitigation. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 16–32, Bellevue, WA, 2012. USENIX.

[32] Ruowen Wang, Peng Ning, Tao Xie, and Quan Chen. Metasymploit: Day-one defense against script-based attacks with security-enhanced symbolic analysis. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 65–80, Washington, D.C., 2013. USENIX.

[33] Xueqiang Wang, Kun Sun, Yuewu Wang, and Jiwu Jing. Deepdroid: Dynamically enforcing enterprise policy on android devices. In *Proc. 22nd Annual Network and Distributed System Security Symposium (NDSS'15). The Internet Society*, 2015.

[34] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.

[35] Yunjing Xu, Michael Bailey, Eric Vander Weele, and Farnam Jahanian. Canvus: Context-aware network vulnerability scanning. In Somesh Jha, Robin Sommer, and Christian Kreibich, editors, *Recent Advances in Intrusion Detection*, volume 6307 of *Lecture Notes in Computer Science*, pages 138–157. Springer Berlin Heidelberg, 2010.

[36] Jason Zander. New windows server containers and azure support for docker. http://azure.microsoft.com/blog/2014/10/15/new-windows-server-containers-and-azure-support-for-docker/?WT.mc_id=Blog_ServerCloud_Announce_TTD, 2014.

[37] Angeliki Zavou, Georgios Portokalidis, and Angelos D. Keromytis. Self-healing multitier architectures using cascading rescue points. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 379–388, New York, NY, USA, 2012. ACM.

[38] Mu Zhang and Heng Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS 2014)*, 2014.

This page intentionally left blank.

This page intentionally left blank.